Contents

[1 Git Started 1](#_Toc527036538)

[2 Git Basics 3](#_Toc527036539)

[3 Git Branching 11](#_Toc527036540)

[4 Git on the Server 16](#_Toc527036541)

[5 Distributed Git 16](#_Toc527036542)

[6 GitHub 20](#_Toc527036543)

# 1 Git Started

The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.

Git comes with a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places

1. /etc/gitconfig file: Contains values applied to every user on the system and all their repositories. If you pass the option --system to git config, it reads and writes from this file specifically. (Because this is a system configuration file, you would need administrative or superuser privilege to make changes to it.)

2. ~/.gitconfig or ~/.config/git/config file: Values specific personally to you, the user. You can make Git read and write to this file specifically by passing the --global option, and this affects *all* of the repositories you work with on your system.

3. config file in the Git directory (that is, .git/config) of whatever repository you’re currently using: Specific to that single repository. You can force Git to read from and write to this file with the --local option, but that is in fact the default. (Unsurprisingly, you need to be located somewhere in a Git repository for this option to work properly.)

Each level overrides values in the previous level

On Windows systems, Git looks for the .gitconfig file in the $HOME directory (C:Users\$USER for most people). It also still looks for /etc/gitconfig, although it’s relative to the MSys root, which is wherever you decide to install Git on your Windows system when you run the installer. If you are using version 2.x or later of Git for Windows, there is also a system-level config file at C:\Documents and Settings\All Users\Application Data\Git\config on Windows XP, and in C:\ProgramData\Git\config on Windows Vista and newer. This config file can only be changed by git config -f <file> as an admin.

**Your Identity**

The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information, and it’s immutably baked into the commits you start creating:

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

**Your Editor**

Now that your identity is set up, you can configure the default text editor that will be used when Git needs you to type in a message. If not configured, Git uses your system’s default editor.If you want to use a different text editor, such as Emacs, you can do the following:

$ git config --global core.editor emacs

If you want to check your configuration settings, you can use the git config --list command to list all the settings Git can find at that point:

You can also check what Git thinks a specific key’s value is by typing git config <key>:

$ git config user.name

Since Git might read the same configuration variable value from more than one file, it’s possible that you have an unexpected value for one of these values and you don’t know why. In cases like that, you can query Git as to the *origin* for that value, and it will tell you which configuration file had the final say in setting that

value:

$ git config --show-origin rerere.autoUpdate

file:/home/johndoe/.gitconfig false

If you ever need help while using Git, there are two equivalent ways to get the comprehensive manual page (manpage) help for any of the Git commands:

$ git help <verb>

In addition, if you don’t need the full-blown manpage help, but just need a quick refresher on the available options for a Git command, you can ask for the more concise “help” output with the -h or --help options, as in:

$ git add –h

# 2 Git Basics

**Short Status**

While the git status output is pretty comprehensive, it’s also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run git status -s or git status --short you get a far more simplified output from the command:

$ git status –s

New files that aren’t tracked have a ?? next to them, new files that have been added to the staging area have an A, modified files have an M and so on. There are two columns to the output - the lefthand column indicates the status of the staging area and the right-hand column indicates the status of the working tree. So for example in that output, the README file is modified in the working directory but not yet staged, while the lib/simplegit.rb file is modified and staged. The Rakefile was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

**Ignoring Files**

Often, you’ll have a class of files that you don’t want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system. In such cases, you can create a file listing patterns to match them named .gitignore. Here is an example .gitignore file:

$ cat .gitignore

\*.[oa]

\*~

The first line tells Git to ignore any files ending in “.o” or “.a” — object and archive files that may be the product of building your code. The second line tells Git to ignore all files whose names end with a tilde (~), which is used by many text editors such as Emacs to mark temporary files. You may also include a log, tmp, or pid directory; automatically generated documentation; and so on. Setting up a .gitignore file for your new repository before you get going is generally a good idea so you don’t accidentally commit files that you really don’t want in your Git repository. The rules for the patterns you can put in the .gitignore file are as follows:

* Blank lines or lines starting with # are ignored.
* Standard glob patterns work, and will be applied recursively throughout the entire working tree.
* You can start patterns with a forward slash (/) to avoid recursivity.
* You can end patterns with a forward slash (/) to specify a directory.
* You can negate a pattern by starting it with an exclamation point (!).

Glob patterns are like simplified regular expressions that shells use. An asterisk (\*) matches zero or more characters; [abc] matches any character inside the brackets (in this case a, b, or c); a question mark (?) matches a single character; and brackets enclosing characters separated by a hyphen ([0-9]) matches any character between them (in this case 0 through 9). You can also use two asterisks to match nested directories; a/\*\*/z would match a/z, a/b/z, a/b/c/z, and so on. Here is another example .gitignore file:

# ignore all .a files

\*.a

# but do track lib.a, even though you're ignoring .a files above

!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO

/TODO

# ignore all files in any directory named build

build/

# ignore doc/notes.txt, but not doc/server/arch.txt

doc/\*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories

doc/\*\*/\*.pdf

**Viewing Your Staged and Unstaged Changes**

To see what you’ve changed but not yet staged, type git diff with no other arguments. That command compares what is in your working directory with what is in your staging area. The result tells you the changes you’ve made that you haven’t yet staged.

If you want to see what you’ve staged that will go into your next commit, you can use git diff --staged. This command compares your staged changes to your last commit:

It’s important to note that git diff by itself doesn’t show all changes made since your last commit — only changes that are still unstaged. If you’ve staged all of your changes, git diff will give you no output.

Now you can use git diff to see what is still unstaged and git diff --cached to see what you’ve staged so far (--staged and --cached are synonyms):

*Git Diff in an External Tool*

We will continue to use the git diff command in various ways throughout the rest of the book. There is another way to look at these diffs if you prefer a graphical or external diff viewing program instead. If you run git difftool instead of git diff, you can view any of these diffs in software like emerge, vimdiff and many more (including commercial products). Run git difftool --tool-help to see what is available on your system

If you want to skip the staging area, Git provides a simple shortcut. Adding the -a option to the git commit command makes Git automatically stage every file that is already tracked before doing the commit,

$ git commit -am 'added new benchmarks'

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The git rm command does that, and also removes the file from your working directory so you don’t see it as an untracked file the next time around.

The next time you commit, the file will be gone and no longer tracked. If you modified the file and added it to the staging area already, you must force the removal with the -f option. This is a safety feature to prevent accidental removal of data that hasn’t yet been recorded in a snapshot and that can’t be recovered from Git.

Another useful thing you may want to do is to keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore. This is particularly useful if you forgot to add something to your .gitignore file and accidentally staged it, like a large log file or a bunch of .a compiled files. To do this, use the --cached option:

$ git rm --cached README

That means you can do things such as:

$ git rm log/\*.log

$ git rm /\*~

**Moving Files**

Unlike many other VCS systems, Git doesn’t explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file. However, Git is pretty smart about figuring that out after the fact — we’ll deal with detecting file movement a bit later. Thus it’s a bit confusing that Git has a mv command. If you want to rename a file in Git, you can run something like:

$ git mv file\_from file\_to

However, this is equivalent to running something like this:

$ mv README.md README

$ git rm README.md

$ git add README

Git figures out that it’s a rename implicitly, so it doesn’t matter if you rename a file that way or with the mv command. The only real difference is that git mv is one command instead of three — it’s a convenience function. More importantly, you can use any tool you like to rename a file, and address

the add/rm later, before you commit.

**Viewing the Commit History**

The most basic and powerful tool to do this is the git log command.

One of the more helpful options is -p or --patch, which shows the difference (the *patch* output) introduced in each commit. You can also limit the number of log entries displayed, such as using -2 to show only the last two entries.

$ git log -p -2

if you want to see some abbreviated stats for each commit, you can use the --stat option

As you can see, the --stat option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end.

Another really useful option is --pretty. This option changes the log output to formats other than the default. A few prebuilt options are available for you to use. The oneline option prints each commit on a single line, which is useful if you’re looking at a lot of commits. In addition, the short, full, and fuller options show the output in roughly the same format but with less or more information, respectively:

$ git log --pretty=oneline

The most interesting option is format, which allows you to specify your own log output format. This is especially useful when you’re generating output for machine parsing — because you specify the format explicitly, you know it won’t change with updates to Git:

$ git log --pretty=format:"%h - %an, %ar : %s"

ca82a6d - Scott Chacon, 6 years ago : changed the version number

085bb3b - Scott Chacon, 6 years ago : removed unnecess

**Option Description of Output**

%H Commit hash

%h Abbreviated commit hash

%T Tree hash

%t Abbreviated tree hash

%P Parent hashes

%p Abbreviated parent hashes

%an Author name

%ae Author email

%ad Author date (format respects the --date=option)

%ar Author date, relative

%cn Committer name

%ce Committer email

%cd Committer date

%cr Committer date, relative

%s Subject

The oneline and format options are particularly useful with another log option called --graph. This option adds a nice little ASCII graph showing your branch and merge history:

*Table 2. Common options to* git log

**Option Description**

-p Show the patch introduced with each commit.

--stat Show statistics for files modified in each commit.

--shortstat Display only the changed/insertions/deletions line from the --stat command.

--name-only Show the list of files modified after the commit information.

--name-status Show the list of files affected with added/modified/deleted information as well.

--abbrev-commit Show only the first few characters of the SHA-1 checksum instead of all 40.

--relative-date Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format.

--graph Display an ASCII graph of the branch and merge history beside the log output.

--pretty Show commits in an alternate format. Options include oneline, short, full,fuller, and format (where you specify your own format).

--oneline Shorthand for --pretty=oneline --abbrev-commit used together.

**Limiting Log Output**

In addition to output-formatting options, git log takes a number of useful limiting options

* -<n>, where n is any integer to show the last n commits
* --since and --until the time-limiting options.

$ git log --since=2.weeks

This command works with lots of formats — you can specify a specific date like "2008-01-15", or a relative date such as "2 years 1 day 3 minutes ago"

* The --author option allows you to filter on a specific author,
* the --grep option lets you search for keywords in the commit messages

Another really helpful filter is the -S option (colloquially referred to as Git’s “pickaxe” option), which takes a string and shows only those commits that changed the number of occurrences of that string.

$ git log -S FindNextStructure

The last really useful option to pass to git log as a filter is a path. If you specify a directory or file name, you can limit the log output to commits that introduced a change to those files. This is always the last option and is generally preceded by double dashes (--) to separate the paths from the options.

**Option Description**

-<n> Show only the last n commits

--since, --after Limit the commits to those made after the specified date.

--until, --before Limit the commits to those made before the specified date.

--author Only show commits in which the author entry matches the specified string.

--committer Only show commits in which the committer entry matches the specified string.

--grep Only show commits with a commit message containing the string

-S Only show commits adding or removing code matching the string

$ git log --pretty="%h - %s" --author='Junio C Hamano' --since="2008-10-01"

--before="2008-11-01" --no-merges -- testData/

To prevent the display of merge commits cluttering up your log history, simply add the log option --no-merges

**Undoing Things**

If you want to redo that commit, make the additional changes you forgot, stage them, and commit again using the --amend option:

$ git commit –amend

**Unstaging a Staged File**

git reset HEAD <file> to unstagea file

**Unmodifying a Modified File**

$ git checkout -- CONTRIBUTING.md

It’s important to understand that git checkout -- <file> is a dangerous command. Any changes you made to that file are gone — Git just copied another file over it. Don’t ever use this command unless you absolutely know that you don’t want the file.

Remember, anything that is *committed* in Git can almost always be recovered. Even commits that were on branches that were deleted or commits that were overwritten with an --amend commit can be recovered (see Data Recovery for data recovery). However, anything you lose that was never committed is likely never to be seen again.

**Working with Remotes**

You can also specify -v, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote:

**Adding Remote Repositories**

To add a new remote Git repository as a shortname you can reference easily, run

$ git remote add <shortname> <url>:

**Fetching and Pulling from Your Remotes**

As you just saw, to get data from your remote projects, you can run:

$ git fetch <remote>

It’s important to note that the git fetch command only downloads the data to your local repository —

git pull command to automatically fetch and then merge that remote branch into your current branch

Running git pull generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you’re currently working on.

**Pushing to Your Remotes**

The command for this is simple: git push <remote> <branch>. If you want to push your master branch to your origin server (again, cloning generally sets up both of those names for you automatically), then you can run this to push any commits you’ve done back up to the server:

$ git push origin master

**Inspecting a Remote**

If you want to see more information about a particular remote, you can use the git remote show <remote> command

$ git remote show origin

This command shows which branch is automatically pushed to when you run git push while on certain branches. It also shows you which remote branches on the server you don’t yet have, which remote branches you have that have been removed from the server, and multiple local branches that are able to merge automatically with their remote-tracking branch when you run git pull.

**Renaming and Removing Remotes**

For instance, if you want to rename pb to paul, you can do so with git remote rename

If you want to remove a remote for some reason — you’ve moved the server or are no longer using a particular mirror, or perhaps a contributor isn’t contributing anymore — you can either use git remote remove or git remote rm:

**Tagging**

**Listing Your Tags**

Listing the available tags in Git is straightforward. Just type git tag (with optional -l or --list):

$ git tag

You can also search for tags that match a particular pattern

$ git tag -l "v1.8.5\*"

If you want just the entire list of tags, running the command git tag implicitly assumes you want a listing and provides one; the use of -l or --list in this case is optional. If, however, you’re supplying a wildcard pattern to match tag names, the use of –l or --list is mandatory.

**Creating Tags**

Git supports two types of tags: *lightweight* and *annotated*.

A lightweight tag is very much like a branch that doesn’t change — it’s just a pointer to a specific commit.

Annotated tags, however, are stored as full objects in the Git database. The easiest way is to specify -a when you run the tag command:

$ git tag -a v1.4 -m "my version 1.4"

The -m specifies a tagging message, which is stored with the tag. If you don’t specify a message for an annotated tag, Git launches your editor so you can type it in.

You can see the tag data along with the commit that was tagged by using the git show command

$ git show v1.4

**Lightweight Tags**

Another way to tag commits is with a lightweight tag. To create a lightweight tag, just provide a tag name:

$ git tag v1.4

**Tagging Later**

To tag that commit, you specify the commit checksum (or part of it) at the end of the command:

$ git tag -a v1.2 9fceb02

**Sharing Tags**

By default, the git push command doesn’t transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches — you can run git push origin <tagname>.

$ git push origin v1.5

you can also use the --tags option to the git push command. This will transfer all of your tags to the remote server that are not already there.

$ git push origin –tags

**Checking out Tags**

If you want to view the versions of files a tag is pointing to, you can do a git checkout

$ git checkout 2.0.0

**Git Aliases**

If you don’t want to type the entire text of each of the Git commands, you can easily set up an alias for each command using git config

$ git config --global alias.unstage 'reset HEAD --'

It’s also common to add a last command, like this:

$ git config --global alias.last 'log -1 HEAD'

However, maybe you want to run an external command, rather than a Git subcommand. In that case, you start the command with a ! character. This is useful if you write your own tools that work with a Git repository. We can demonstrate by aliasing git visual to run gitk:

$ git config --global alias.visual '!gitk'

# 3 Git Branching

How does Git know what branch you’re currently on? It keeps a special pointer called HEAD. Note that this is a lot different than the concept of HEAD in other VCSs you may be used to, such as Subversion or CVS. In Git, this is a pointer to the local branch you’re currently on.

To switch to an existing branch, you run the git checkout command. Let’s switch to the new testing branch:

$ git checkout testing

This moves HEAD to point to the testing branch.

That command did two things. It moved the HEAD pointer back to point to the master branch, and it reverted the files in your working directory back to the snapshot that master points to. This also means the changes you make from this point forward will diverge from an older version of the project. It essentially rewinds the work you’ve done in your testing branch so you can go in a different direction.

Because a branch in Git is actually a simple file that contains the 40 character SHA-1 checksum of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline).

This resolution has a little of each section, and the <<<<<<<, =======, and >>>>>>> lines have been completely removed. After you’ve resolved each of these sections in each conflicted file, run git add on each file to mark it as resolved. Staging the file marks it as resolved in Git.

If you want to use a graphical tool to resolve these issues, you can run git mergetool, which fires up an appropriate visual merge tool and walks you through the conflicts.

You can run git status again to verify that all conflicts have been resolved:

**Branch Management**

To see the last commit on each branch, you can run git branch –v:

$ git branch –v

The useful --merged and --no-merged options can filter this list to branches that you have or have not yet merged into the branch you’re currently on. To see which branches are already merged into the branch you’re on, you can run git branch --merged:

$ git branch --merged

To see all the branches that contain work you haven’t yet merged in, you can run git branch --no-merged:

The options described above, --merged and --no-merged will, if not given a commit or branch name as an argument, show you what is, respectively, merged or not merged into your *current* branch.

You can always provide an additional argument to ask about the merge state with respect to some other branch without checking that other branch out first, as in, what is not merged into the master branch?

$ git checkout testing

$ git branch --no-merged master

**Remote Branches**

You can get a full list of remote references explicitly with git ls-remote [remote], or git remote show [remote] for remote branches as well as more information.

Remote-tracking branches are references to the state of remote branches. They’re local references that you can’t move; Git moves them for you whenever you do any network communication, to make sure they accurately represent the state of the remote repository. Think of them as bookmarks, to remind you where the branches in your remote repositories were the last time you connected to them.

Remote-tracking branches take the form <remote>/<branch>.

*“origin” is not special*

Just like the branch name “master” does not have any special meaning in Git, neither does “origin”. While “master” is the default name for a starting branch when you run git init which is the only reason it’s widely used, “origin” is the default name for a remote when you run git clone. If you run git clone -o booyah instead, then you will have booyah/master as your default remote branch.

To synchronize your work, you run a git fetch origin command. This command looks up which server “origin” is (in this case, it’s git.ourcompany.com), fetches any data from it that you don’t yet have, and updates your local database, moving your origin/master pointer to its new, more up-todate position.

**Pushing**

When you want to share a branch with the world, you need to push it up to a remote that you have write access to. Your local branches aren’t automatically synchronized to the remotes you write to — you have to explicitly push the branches you want to share. That way, you can use private branches for work you don’t want to share, and push up only the topic branches you want to collaborate on.

If you have a branch named serverfix that you want to work on with others, you can push it up the same way you pushed your first branch. Run git push <remote> <branch>:

$ git push origin serverfix

This is a bit of a shortcut. Git automatically expands the serverfix branchname out to refs/heads/serverfix:refs/heads/serverfix, which means, “Take my serverfix local branch and push it to update the remote’s serverfix branch.”

You can also do git push origin serverfix:serverfix, which does the same thing — it says, “Take my serverfix and make it the remote’s serverfix.” You can use this format to push a local branch into a remote branch that is named differently. If you didn’t want it to be called serverfix on the remote, you could instead run git push origin serverfix:awesomebranch to push your local serverfix branch to the awesomebranch branch on the remote project.

It’s important to note that when you do a fetch that brings down new remote-tracking branches, you don’t automatically have local, editable copies of them. In other words, in this case, you don’t have a new serverfix branch — you only have an origin/serverfix pointer that you can’t modify

To merge this work into your current working branch, you can run git merge origin/serverfix. If you want your own serverfix branch that you can work on, you can base it off your remotetracking branch:

$ git checkout -b serverfix origin/serverfix

This gives you a local branch that you can work on that starts where origin/serverfix is.

**Tracking Branches**

Checking out a local branch from a remote-tracking branch automatically creates what is called a “tracking branch” (and the branch it tracks is called an “upstream branch”). Tracking branches are local branches that have a direct relationship to a remote branch. If you’re on a tracking branch and type git pull, Git automatically knows which server to fetch from and which branch to merge in.

When you clone a repository, it generally automatically creates a master branch that tracks origin/master. However, you can set up other tracking branches if you wish — ones that track branches on other remotes, or don’t track the master branch. The simple case is the example you just saw, running git checkout -b <branch> <remote>/<branch>. This is a common enough operation that Git provides the --track shorthand:

$ git checkout --track origin/serverfix

In fact, this is so common that there’s even a shortcut for that shortcut. If the branch name you’re trying to checkout (a) doesn’t exist and (b) exactly matches a name on only one remote, Git will create a tracking branch for you:

$ git checkout serverfix

To set up a local branch with a different name than the remote branch, you can easily use the first version with a different local branch name:

$ git checkout -b sf origin/serverfix

Now, your local branch sf will automatically pull from origin/serverfix.

If you already have a local branch and want to set it to a remote branch you just pulled down, or want to change the upstream branch you’re tracking, you can use the -u or --set-upstream-to option to git branch to explicitly set it at any time.

$ git branch -u origin/serverfix

*Upstream shorthand*

When you have a tracking branch set up, you can reference its upstream branch with the @{upstream} or @{u} shorthand. So if you’re on the master branch and it’s tracking origin/master, you can say something like git merge @{u} instead of git merge origin/master if you wish.

If you want to see what tracking branches you have set up, you can use the -vv option to git branch. This will list out your local branches with more information including what each branch is tracking and if your local branch is ahead, behind or both.

**Pulling**

Generally it’s better to simply use the fetch and merge commands explicitly as the magic of git pull can often be confusing

**Deleting Remote Branches**

You can delete a remote branch using the --delete option to git push. If you want to delete your serverfix branch from the server, you run the following

$ git push origin --delete serverfix

**Rebasing**

In Git, there are two main ways to integrate changes from one branch into another: the merge and the rebase.

It works by going to the common ancestor of the two branches (the one you’re on and the one you’re rebasing onto), getting the diff introduced by each commit of the branch you’re on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.

There is no difference in the end product of the integration, but rebasing makes for

a cleaner history. If you examine the log of a rebased branch, it looks like a linear history: it appears that all the work happened in series, even when it originally happened in parallel

Note that the snapshot pointed to by the final commit you end up with, whether it’s the last of the rebased commits for a rebase or the final merge commit after a merge, is the same snapshot – it’s only the history that is different. Rebasing replays changes from one line of work onto another in the order they were introduced, whereas merging takes the endpoints and merges them together

**More Interesting Rebases**

****

Suppose you decide that you want to merge your client-side changes into your mainline for a release, but you want to hold off on the server-side changes until it’s tested further. You can take the changes on client that aren’t on server (C8 and C9) and replay them on your master branch by using the --onto option of git rebase:

$ git rebase --onto master server client

This basically says, “Take the client branch, figure out the patches since it diverged from the server branch, and replay these patches in the client branch as if it was based directly off the master branch instead.” It’s a bit complex, but the result is pretty cool.

Let’s say you decide to pull in your server branch as well. You can rebase the server branch onto the master branch without having to check it out first by running git rebase <basebranch> <topicbranch> — which checks out the topic branch (in this case, server) for you and replays it onto the base branch (master):

$ git rebase master server

# 4 Git on the Server

**GitLab**

https://bitnami.com/stack/gitlab,

# 5 Distributed Git

The Git project has well-formatted commit messages — try running git log --no-merges there to see what a nicely-formatted project-commit history looks like.

The main difference is that merges happen client-side rather than on the server at commit time.

This is especially important to understand if you’re used to Subversion, because you’ll notice that the two developers didn’t edit the same file. Although Subversion automatically does such a merge on the server if different files are edited, with Git, you must *first* merge the commits locally. In other words, John must first fetch Jessica’s upstream changes and merge them into his local repository before he will

be allowed to push.

$ git log --no-merges issue54..origin/master

The issue54..origin/master syntax is a log filter that asks Git to display only those commits that are on the latter branch (in this case origin/master) that are not on the first branch (in this case issue54). We’ll go over this syntax in detail in Commit Ranges.

At this point, Jessica wants to push all of this merged “featureB” work back to the server, but she doesn’t want to simply push her own featureB branch. Rather, since Josie has already started an upstream featureBee branch, Jessica wants to push to *that* branch, which she does with:

$ git push -u origin featureB:featureBe

This is called a *refspec*. See The Refspec for a more detailed discussion of Git refspecs and different things you can do with them. Also notice the -u flag; this is short for --set-upstream, whichconfigures the branches for easier pushing and pulling later.

**Forked Public Project**

$ git clone <url>

$ cd project

$ git checkout -b featureA

... work ...

$ git commit

... work ...

$ git commit

……..

$ git remote add myfork <url>

$ git push -u myfork featureA

You start a new branch based off the current origin/master branch, squash the featureB changes there, resolve any conflicts, make the implementation change, and then push that as a new branch:

$ git checkout -b featureBv2 origin/master

$ git merge --squash featureB

... change implementation ...

$ git commit

$ git push myfork featureBv2



The --squash option takes all the work on the merged branch and squashes it into one changeset producing the repository state as if a real merge happened, without actually making a merge commit. This means your future commit will have one parent only and allows you to introduce all the changes from another branch and then make more changes before recording the new commit. Also the --no-commit option can be useful to delay the merge commit in case of the default merge process.

At this point, you can notify the maintainer that you’ve made the requested changes, and that they can find those changes in your featureBv2 branch.

**Determining What Is Introduced**

It’s often helpful to get a review of all the commits that are in this branch but that aren’t in your master branch. You can exclude commits in the master branch by adding the --not option before the branch name. This does the same thing as the master..contrib format that we used earlier. For example, if your contributor sends you two patches and you create a branch called contrib and applied those patches there, you can run this:

$ git log contrib --not master

To see what changes each commit introduces, remember that you can pass the -p option to git log and it will append the diff introduced to each commit.

To see a full diff of what would happen if you were to merge this topic branch with another branch, you may have to use a weird trick to get the correct results. You may think to run this:

$ git diff master

This command gives you a diff, but it may be misleading. If your master branch has moved forward since you created the topic branch from it, then you’ll get seemingly strange results. This happens because Git directly compares the snapshots of the last commit of the topic branch you’re on and the snapshot of the last commit on the master branch. For example, if you’ve added a line in a file on the master branch, a direct comparison of the snapshots will look like the topic branch is going to remove that line.

If master is a direct ancestor of your topic branch, this isn’t a problem; but if the two histories have diverged, the diff will look like you’re adding all the new stuff in your topic branch and removing everything unique to the master branch.

What you really want to see are the changes added to the topic branch — the work you’ll introduce if you merge this branch with master. You do that by having Git compare the last commit on your topic branch with the first common ancestor it has with the master branch.

Technically, you can do that by explicitly figuring out the common ancestor and then running your diff on it:

$ git merge-base contrib master

36c7dba2c95e6bbb78dfa822519ecfec6e1ca649

$ git diff 36c7db

or, more concisely:

$ git diff $(git merge-base contrib master)

However, neither of those is particularly convenient, so Git provides another shorthand for doing the same thing: the triple-dot syntax. In the context of the git diff command, you can put three periods after another branch to do a diff between the last commit of the branch you’re on and its common ancestor with another branch:

$ git diff master...contrib

This command shows you only the work your current topic branch has introduced since its common ancestor with master. That is a very useful syntax to remember.

**Rerere**

If you’re doing lots of merging and rebasing, or you’re maintaining a long-lived topic branch, Git has a feature called “rerere” that can help. Rerere stands for “reuse recorded resolution” — it’s a way of shortcutting manual conflict resolution. When rerere is enabled, Git will keep a set of pre- and post-images from successful merges, and if it notices that there’s a conflict that looks exactly like one you’ve already fixed, it’ll just use the fix from last time, without bothering you with it.

This feature comes in two parts: a configuration setting and a command. The configuration setting is rerere.enabled, and it’s handy enough to put in your global config:

$ git config --global rerere.enabled true

**Generating a Build Number**

Because Git doesn’t have monotonically increasing numbers like *v123* or the equivalent to go with each commit, if you want to have a human-readable name to go with a commit, you can run git describe on that commit. Git gives you the name of the nearest tag with the number of commits on top of that tag and a partial SHA-1 value of the commit you’re describing:

$ git describe master

This way, you can export a snapshot or build and name it something understandable to people. In fact, if you build Git from source code cloned from the Git repository, git --version gives you something that looks like this. If you’re describing a commit that you have directly tagged, it gives you the tag name.

The git describe command favors annotated tags (tags created with the -a or -s flag), so release tags should be created this way if you’re using git describe, to ensure the commit is named properly when described. You can also use this string as the target of a checkout or show command, although it relies on the abbreviated SHA-1 value at the end, so it may not be valid forever. For instance, the Linux kernel recently jumped from 8 to 10 characters to ensure SHA-1 object uniqueness, so older git describe output names were invalidated.

**Preparing a Release**

Now you want to release a build. One of the things you’ll want to do is create an archive of the latest snapshot of your code for those poor souls who don’t use Git. The command to do this is git archive:

$ git archive master --prefix='projet/' | gzip > `git describe master`.tar.gz

$ ls \*.tar.gz

v1.6.2-rc1-20-g8c5b85c.tar.gz

$ git archive master --prefix='project/' --format=zip > `git describe master`.zip

**Shortlog**

It’s time to email your mailing list of people who want to know what’s happening in your project. A nice way of quickly getting a sort of changelog of what has been added to your project since your last release or email is to use the git shortlog command. It summarizes all the commits in the range you give it; for example, the following gives you a summary of all the commits since your last release, if your last release was named v1.0.1:

$ git shortlog --no-merges master --not v1.0.1

# 6 GitHub

To demonstrate this, we’re going to use a low-level command (often referred to as a “plumbing”command, which we’ll read about more in Plumbing and Porcelain) called ls-remote. This command is generally not used in day-to-day Git operations but it’s useful to show us what references are present on the server.

If we run this command against the “blink” repository we were using earlier, we will get a list of all the branches and tags and other references in the repository.

# 7 Git Tools

Git can figure out a short, unique abbreviation for your SHA-1 values. If you pass --abbrev-commit to the git log command, the output will use shorter values but keep them unique; it defaults to using seven characters but makes them longer if necessary to keep the SHA-1 unambiguous:

$ git log –abrev-commit –pretty=oneline

If you want to see which specific SHA-1 a branch points to, or if you want to see what any of these examples boils down to in terms of SHA-1s, you can use a Git plumbing tool called rev-parse

$ git rev-parse topic1

One of the things Git does in the background while you’re working away is keep a “reflog” — a log of where your HEAD and branch references have been for the last few months. You can see your reflog by using git reflog:

If you want to see the fifth prior value of the HEAD of your repository, you can use the @{5} reference that you see in the reflog output:

$ git show HEAD@{5}

To see reflog information formatted like the git log output, you can run git log -g

It’s important to note that reflog information is strictly local — it’s a log only of what *you’ve* done in *your* repository.

**Ancestry References**

The other main way to specify a commit is via its ancestry. If you place a ^ (caret) at the end of a reference, Git resolves it to mean the parent of that commit

Then, you can see the previous commit by specifying HEAD^, which means “the parent of HEAD”:

You can also specify a number after the ^ – for example, d921970^2 means “the second parent of d921970.” This syntax is useful only for merge commits, which have more than one parent. The first parent is the branch you were on when you merged, and the second is the commit on the branch that you merged in:

The other main ancestry specification is the ~ (tilde). This also refers to the first parent, so HEAD~ and HEAD^ are equivalent. The difference becomes apparent when you specify a number. HEAD~2 means “the first parent of the first parent,” or “the grandparent” — it traverses the first parents the number of times you specify. For example, in the history listed earlier.

HEAD~3 peut aussi s’ecrire HEAD^^^.

You can also combine these syntaxes — you can get the second parent of the previous reference (assuming it was a merge commit) by using HEAD~3^2, and so on

**Commit Ranges**

**Double Dot**

Say you want to see what is in your experiment branch that hasn’t yet been merged into your master branch. You can ask Git to show you a log of just those commits with master..experiment — that means “all commits reachable from experiment that aren’t reachable from master.” For the sake of brevity and clarity in these examples, the letters of the commit objects from the diagram are used in place of the actual log output in the order that they would display:

$ git log master..experiment

$ git log origin/master..HEAD

This command shows you any commits in your current branch that aren’t in the master branch on your origin remote. If you run a git push and your current branch is tracking origin/master, the commits listed by git log origin/master..HEAD are the commits that will be transferred to the server. You can also leave off one side of the syntax to have Git assume HEAD. For example, you can get the same results as in the previous example by typing git log origin/master.. — Git substitutes HEAD if one side is missing.

**Multiple Points**

The double-dot syntax is useful as a shorthand, but perhaps you want to specify more than two branches to indicate your revision, such as seeing what commits are in any of several branches that aren’t in the branch you’re currently on. Git allows you to do this by using either the ^ character or --not before any reference from which you don’t want to see reachable commits. Thus, the following three commands are equivalent:

$ git log refA..refB

$ git log ^refA refB

$ git log refB --not refA

This is nice because with this syntax you can specify more than two references in your query, which you cannot do with the double-dot syntax. For instance, if you want to see all commits that are reachable from refA or refB but not from refC, you can use either of:

$ git log refA refB ^refC

$ git log refA refB --not refC

**Triple Dot**

The last major range-selection syntax is the triple-dot syntax, which specifies all the commits that are reachable by *either* of two references but not by both of them

$ git log master...experiment

A common switch to use with the log command in this case is --left-right, which shows you which side of the range each commit is in. This helps make the output more useful

$ git log --left-right master...experiment

With these tools, you can much more easily let Git know what commit or commits you want to inspect.

**Interactive Staging**

If you run git add with the -i or --interactive option, Git enters an interactive shell mode

Patch Mode

$ git add –p

$ git reset –p

$ git checkout –p

$ git stash save -p

**Stashing and Cleaning**

Stashing takes the dirty state of your working directory — that is, your modified tracked files and staged changes — and saves it on a stack of unfinished changes that you can reapply at any time (even on a different branch).

$ git stash save

$ git stash list

*Migrating to* git stash push

As of late October 2017, there has been extensive discussion on the Git mailing list,

wherein the command git stash save is being deprecated in favour of the existing

alternative git stash push. The main reason for this is that git stash push

introduces the option of stashing selected *pathspecs*, something git stash save

does not support

To push a new stash onto your stack, run git stash or git stash save:

To see which stashes you’ve stored, you can use git stash list

You can reapply the one you just stashed by using the command shown in the help output of the original stash command: git stash apply. If you want to apply one of the older stashes, you can specify it by naming it, like this: git stash apply stash@{2}. If you don’t specify a stash, Git assumes the most recent stash and tries to apply it:

$ git stash apply

The changes to your files were reapplied, but the file you staged before wasn’t restaged. To do that, you must run the git stash apply command with a --index option to tell the command to try to reapply the staged changes.

To remove it, you can run git stash drop with the name of the stash to remove. You can also run git stash pop to apply the stash and then immediately drop it from your stack.

**Creative Stashing**

There are a few stash variants that may also be helpful. The first option that is quite popular is the --keep-index option to the stash save command. This tells Git to not only include all staged content in the stash being created, but simultaneously leave it in the index.

Another common thing you may want to do with stash is to stash the untracked files as well as the tracked ones. By default, git stash will stash only modified and staged *tracked* files. If you specify --include-untracked or -u, Git will include untracked files in the stash being created.

Finally, if you specify the --patch flag, Git will not stash everything that is modified but will instead prompt you interactively which of the changes you would like to stash and which you would like to keep in your working directory.

**Creating a Branch from a Stash**

If you stash some work, leave it there for a while, and continue on the branch from which you stashed the work, you may have a problem reapplying the work. If the apply tries to modify a file that you’ve since modified, you’ll get a merge conflict and will have to try to resolve it. If you want an easier way to test the stashed changes again, you can run git stash branch <branch>, which creates a new branch for you with your selected branch name, checks out the commit you were on when you stashed your work, reapplies your work there, and then drops the stash if it applies successfully:

$ git stash branch testchanges

**Cleaning your Working Directory**

You’ll want to be pretty careful with this git clean command, since it’s designed to remove files from your working directory that are not tracked. If you change your mind, there is often no retrieving the content of those files. A safer option is to run git stash --all to remove everything but save it in a stash.

Assuming you do want to remove cruft files or clean your working directory, you can do so with git clean. To remove all the untracked files in your working directory, you can run git clean -f -d, which removes any files and also any subdirectories that become empty as a result. The -f means *force* or "really do this".

If you ever want to see what it would do, you can run the command with the -n option, which means “do a dry run and tell me what you *would* have removed”

$ git clean -d -n

Si vous souhaitez visualiser ce qui serait fait, vous pouvez lancer la commande avec l’option -n qui signifie ≪ fais-le a blanc et montre-moi ce qui *serait* supprime ≫.

$ git clean -d –n

By default, the git clean command will only remove untracked files that are not ignored. Any file that matches a pattern in your .gitignore or other ignore files will not be removed. If you want to remove those files too, such as to remove all .o files generated from a build so you can do a fully clean build, you can add a -x to the clean command.

If you don’t know what the git clean command is going to do, always run it with a -n first to double check before changing the -n to a -f and doing it for real. The other way you can be careful about the process is to run it with the -i or “interactive” flag.

This will run the clean command in an interactive mode.

$ git clean -x –i

**Searching**

Git provides a couple of useful tools for looking through the code and commits stored in its database quickly and easily. We’ll go through a few of them

**Git Grep**

Git ships with a command called grep that allows you to easily search through any committed tree,the working directory, or even the index for a string or regular expression.

By default, git grep will look through the files in your working directory. As a first variation, you can use either of the -n or --line-number options to print out the line numbers where Git has found matches:

$ git grep -n gmtime\_r

In addition to the basic search shown above, git grep supports a plethora of other interesting options.

For instance, instead of printing all of the matches, you can ask git grep to summarize the output by showing you only which files contained the search string and how many matches there were in each file with the -c or --count option:

$ git grep --count gmtime\_r

If you’re interested in the *context* of a search string, you can display the enclosing method or function for each matching string with either of the -p or --show-function options

$ git grep -p gmtime\_r \*.c

You can also search for complex combinations of strings with the --and flag, which ensures that multiple matches must occur in the same line of text. For instance, let’s look for any lines that define a constant whose name contains *either* of the substrings “LINK” or “BUF\_MAX”, specifically in an older version of the Git codebase represented by the tag v1.8.0 (we’ll throw in the --break and --heading options which help split up the output into a more readable format):

$ git grep --break --heading -n -e '#define' --and ( -e LINK -e BUF\_MAX ) v1.8.0

The git grep command has a few advantages over normal searching commands like grep and ack. The first is that it’s really fast, the second is that you can search through any tree in Git, not just the working directory. As we saw in the above example, we looked for terms in an older version of the Git source code, not the version that was currently checked out.

**Git Log Searching**

Perhaps you’re looking not for *where* a term exists, but *when* it existed or was introduced. The git log command has a number of powerful tools for finding specific commits by the content of their messages or even the content of the diff they introduce.

If, for example, we want to find out when the ZLIB\_BUF\_MAX constant was originally introduced, we can use the -S option (colloquially referred to as the Git “pickaxe” option) to tell Git to show us only those commits that changed the number of occurrences of that string.

$ git log -S ZLIB\_BUF\_MAX –oneline

If you need to be more specific, you can provide a regular expression to search for with the –G option.

**Line Log Search**

Another fairly advanced log search that is insanely useful is the line history search. Simply run git log with the -L option, and it will show you the history of a function or line of code in your codebase.

For example, if we wanted to see every change made to the function git\_deflate\_bound in the zlib.c file, we could run git log -L :git\_deflate\_bound:zlib.c. This will try to figure out what the bounds of that function are and then look through the history and show us every change that was made to the function as a series of patches back to when the function was first created.

$ git log -L :git\_deflate\_bound:zlib.c

Git can’t figure out how to match a function or method in your programming language, you can also provide it with a regular expression (or *regex*). For example, this would

have done the same thing as the example above:

$ git log -L '/unsigned long git\_deflate\_bound/',/^}/:zlib.c.

You could also give it a range of lines or a single line number and you’ll get the same sort of output

**Rewriting History**

Many times, when working with Git, you may want to revise your local commit history. This can involve changing the order of the commits, changing messages or modifying files in a commit, squashing together or splitting apart commits, or removing commits entirely — all before you share your

work with others.

One of the cardinal rules of Git is that, since so much work is local within your clone, you have a great deal of freedom to rewrite your history *locally*. However, once you push your work, it is a different story entirely, and you should consider pushed work as final unless you have good reason to change it. In short, you should avoid pushing your work until you’re happy with it and ready to share it with the rest of the world.

**Changing the Last Commit**

$ git commit –amend

The command above loads the previous commit message into an editor session, where you can make changes to the message, save those changes and exit. When you save and close the editor, the editor writes a new commit containing that updated commit message and makes it your new last commit.

If, on the other hand, you want to change the actual *content* of your last commit, the process works basically the same way — first make the changes you think you forgot, stage those changes, and the subsequent git commit --amend *replaces* that last commit with your new, improved commit.

You need to be careful with this technique because amending changes the SHA-1 of the commit. It’s like a very small rebase — don’t amend your last commit if you’ve already pushed it.

you can simply make the changes, stage them, and avoid the unnecessary editor

session entirely with

$ git commit --amend --no-edit

**Changing Multiple Commit Messages**

For example, if you want to change the last three commit messages, or any of the commit messages in that group, you supply as an argument to git rebase -i the parent of the last commit you want to edit, which is HEAD~2^ or HEAD~3. It may be easier to remember the ~3 because you’re trying to edit the last three commits, but keep in mind that you’re actually designating four commits ago, the parent of the last commit you want to edit:

$ git rebase -i HEAD~3

**Reordering Commits**

You can also use interactive rebases to reorder or remove commits entirely. If you want to remove the “added cat-file” commit and change the order in which the other two commits are introduced, you can change the rebase script from this

**Squashing Commits**

It’s also possible to take a series of commits and squash them down into a single commit with the interactive rebasing tool.

**Splitting a Commit**

Splitting a commit undoes a commit and then partially stages and commits as many times as commits you want to end up with.

There, you can do a mixed reset of that commit with git reset HEAD^, which effectively undoes that commit and leaves the modified files unstaged. Now you can stage and commit files until you have several commits, and run git rebase –continue when you’re done:

$ git reset HEAD^

$ git add README

$ git commit -m 'updated README formatting'

$ git add lib/simplegit.rb

$ git commit -m 'added blame'

$ git rebase --continue

**The Nuclear Option: filter-branch**

There is another history-rewriting option that you can use if you need to rewrite a larger number of commits in some scriptable way – for instance, changing your email address globally or removing a file from every commit. The command is filter-branch, and it can rewrite huge swaths of your history, so you probably shouldn’t use it unless your project isn’t yet public and other people haven’t based work off the commits you’re about to rewrite. However, it can be very useful.

**Removing a File from Every Commit**

To remove a file named passwords.txt from your entire history, you can use the --tree-filter option to filter-branch:

$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD

Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)

Ref 'refs/heads/master' was rewritten

The --tree-filter option runs the specified command after each checkout of the project and then recommits the results. In this case, you remove a file called passwords.txt from every snapshot, whether it exists or not. If you want to remove all accidentally committed editor backup files, you can run something like

$ git filter-branch --tree-filter 'rm -f \*~'HEAD.

To run filter-branch on all your branches, you can pass --all to the command

**Reset Demystified**

These commands are two of the most confusing parts of Git when you first encounter them. They do so many things that it seems hopeless to actually understand them and employ them properly.

Git as a system manages and manipulates three trees in its normal operation:

**Tree Role**

HEAD Last commit snapshot, next parent

Index Proposed next commit snapshot

Working Directory Sandbox

**The HEAD**

HEAD is the pointer to the current branch reference, which is in turn a pointer to the last commit made on that branch. That means HEAD will be the parent of the next commit that is created. It’s generally simplest to think of HEAD as the snapshot of **your last commit on that branch**..

$ git cat-file -p HEAD

$ git ls-tree -r HEAD

The Git cat-file and ls-tree commands are “plumbing” commands that are used for lower level things and not really used in day-to-day work, but they help us see what’s going on here.

**The Index**

The Index is your **proposed next commit**. We’ve also been referring to this concept as Git’s “Staging Area” as this is what Git looks at when you run git commit.

Git populates this index with a list of all the file contents that were last checked out into your working directory and what they looked like when they were originally checked out. You then replace some of those files with new versions of them, and git commit converts that into the tree for a new commit.

$ git ls-files –s

Again, here we’re using git ls-files, which is more of a behind the scenes command that shows you what your index currently looks like.

**The Working Directory**

Finally, you have your working directory. The other two trees store their content in an efficient but

inconvenient manner, inside the .git folder. The Working Directory unpacks them into actual files, which makes it much easier for you to edit them. Think of the Working Directory as a **sandbox**, where you can try changes out before committing them to your staging area (index) and then to history.

**The Workflow**

Git’s main purpose is to record snapshots of your project in successively better states, by manipulating these three trees.



Switching branches or cloning goes through a similar process. When you checkout a branch, it changes **HEAD** to point to the new branch ref, populates your **Index** with the snapshot of that commit, then copies the contents of the **Index** into your **Working Directory**.

**The Role of Reset**

The reset command makes more sense when viewed in this context.

Let’s now walk through exactly what reset does when you call it. It directly manipulates these three trees in a simple and predictable way. It does up to three basic operations.

**Step 1: Move HEAD**

The first thing reset will do is move what HEAD points to. This isn’t the same as changing HEAD itself (which is what checkout does); reset moves the branch that HEAD is pointing to.

No matter what form of reset with a commit you invoke, this is the first thing it will always try to do. With reset --soft, it will simply stop there.

**Step 2: Updating the Index (--mixed))**

The next thing reset will do is to update the Index with the contents of whatever snapshot HEAD now points to.

If you specify the --mixed option, reset will stop at this point. This is also the default, so if you specify no option at all (just git reset HEAD~ in this case), this is where the command will stop.

**Step 3: Updating the Working Directory (--hard)**

The third thing that reset will do is to make the Working Directory look like the Index. If you use the --hard option, it will continue to this stage.

It’s important to note that this flag (--hard) is the only way to make the reset command dangerous, and one of the very few cases where Git will actually destroy data. Any other invocation of reset can be pretty easily undone, but the --hard option cannot

**Recap**

The reset command overwrites these three trees in a specific order, stopping when you tell it to:

1. Move the branch HEAD points to *(stop here if* --soft*)*

2. Make the Index look like HEAD *(stop here unless* --hard*)*

3. Make the Working Directory look like the Index

**Reset With a Path**

So, assume we run git reset file.txt. This form (since you did not specify a commit SHA-1 or branch, and you didn’t specify --soft or --hard) is shorthand for git reset --mixed HEAD file.txt,

So it essentially just copies file.txt from HEAD to the Index.

This has the practical effect of *unstaging* the file.

This is why the output of the git status command suggests that you run this to unstage a file. (SeeUnstaging a Staged File for more on this.)

We could just as easily not let Git assume we meant “pull the data from HEAD” by specifying a specific commit to pull that file version from. We would just run something like git reset eb43bf file.txt.

**Squashing**

You can run git reset --soft HEAD~2 to move the HEAD branch back to an older commit (the mostrecent commit you want to keep):

**Check It Out**

Finally, you may wonder what the difference between checkout and reset is. Like reset, checkout manipulates the three trees, and it is a bit different depending on whether you give the command a file path or not.

**Without Paths**

Running git checkout [branch] is pretty similar to running git reset --hard [branch] in that it updates all three trees for you to look like [branch], but there are two important differences.

First, unlike reset --hard, checkout is working-directory safe; it will check to make sure it’s not blowing away files that have changes to them. Actually, it’s a bit smarter than that — it tries to do a trivial merge in the Working Directory, so all of the files you *haven’t* changed will be updated. Reset --hard, on the other hand, will simply replace everything across the board without checking.

The second important difference is how checkout updates HEAD. Whereas reset will move the branch that HEAD points to, checkout will move HEAD itself to point to another branch.



**With Paths**

The other way to run checkout is with a file path, which, like reset, does not move HEAD. It is just like git reset [branch] file in that it updates the index with that file at that commit, but it also overwrites the file in the working directory. It would be exactly like git reset --hard [branch] file (if reset would let you run that) — it’s not working-directory safe, and it does not move HEAD.

Also, like git reset and git add, checkout will accept a --patch option to allow you to selectively revert file contents on a hunk-by-hunk basis.

**Summary**

Here’s a cheat-sheet for which commands affect which trees. The “HEAD” column reads “REF” if that command moves the reference (branch) that HEAD points to, and “HEAD” if it moves HEAD itself. Pay especial attention to the *WD Safe?* column — if it says **NO**, take a second to think before running that command.



**Advanced Merging**

**Merge Conflicts**

**Ignoring Whitespace**

The default merge strategy can take arguments though, and a few of them are about properly ignoring whitespace changes. If you see that you have a lot of whitespace issues in a merge, you can simply abort it and do it again, this time with -Xignore-all-space or -Xignore-space-change. The first option ignores whitespace **completely** when comparing lines, the second treats sequences of one or more whitespace characters as equivalent

$ git merge -Xignore-space-change branch1

First, we get into the merge conflict state. Then we want to get copies of my version of the file, their version (from the branch we’re merging in) and the common version (from where both sides branched off). Then we want to fix up either their side or our side and re-try the merge again for just this single file.

Getting the three file versions is actually pretty easy. Git stores all of these versions in the index under “stages” which each have numbers associated with them. Stage 1 is the common ancestor, stage 2 is your version and stage 3 is from the MERGE\_HEAD, the version you’re merging in (“theirs”).

You can extract a copy of each of these versions of the conflicted file with the git show command and a special syntax.

$ git show :1:hello.rb > hello.common.rb

$ git show :2:hello.rb > hello.ours.rb

$ git show :3:hello.rb > hello.theirs.rb

If you want to get a little more hard core, you can also use the ls-files -u plumbing command to get the actual SHA-1s of the Git blobs for each of these files.

$ git ls-files -u

The :1:hello.rb is just a shorthand for looking up that blob SHA-1.

Now that we have the content of all three stages in our working directory, we can manually fix up theirs to fix the whitespace issue and re-merge the file with the little-known git merge-file command which does just that.

$ git merge-file -p hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

If you want to get an idea before finalizing this commit about what was actually changed between one side or the other, you can ask git diff to compare what is in your working directory that you’re about to commit as the result of the merge to any of these stages. Let’s go through them all.

To compare your result to what you had in your branch before the merge, in other words, to see what the merge introduced, you can run git diff –ours

If we want to see how the result of the merge differed from what was on their side, you can run git diff --theirs.

Finally, you can see how the file has changed from both sides with git diff --base.

**Checking Out Conflicts**

If we open up the file, we’ll see something like this:

#! /usr/bin/env ruby

def hello

<<<<<<< HEAD

puts 'hola world'

======

puts 'hello mundo'

>>>>>>> mundo

end

hello()

Perhaps it’s not obvious how exactly you should fix this conflict. You need more

context.

One helpful tool is git checkout with the ‘--conflict’ option. This will re-checkout the file again and replace the merge conflict markers. This can be useful if you want to reset the markers and try to resolve them again.

You can pass --conflict either diff3 or merge (which is the default). If you pass it diff3, Git will use a slightly different version of conflict markers, not only giving you the “ours” and “theirs” versions, but also the “base” version inline to give you more context.

$ git checkout --conflict=diff3 hello.rb

Une fois que nous l’avons lance, le fichier ressemble a ceci :

#! /usr/bin/env ruby

def hello

<<<<<<< ours

puts 'hola world'

||||||| base

puts 'hello world'

======

puts 'hello mundo'

>>>>>>> theirs

end

hello()

If you like this format, you can set it as the default for future merge conflicts by setting the merge.conflictstyle setting to diff3..

$ git config --global merge.conflictstyle diff3

The git checkout command can also take --ours and --theirs options, which can be a really fast way of just choosing either one side or the other without merging things at all.

This can be particularly useful for conflicts of binary files where you can simply choose one side, or where you only want to merge certain files in from another branch - you can do the merge and then checkout certain files from one side or the other before committing

**Merge Log**

Another useful tool when resolving merge conflicts is git log. This can help you get context on what may have contributed to the conflicts. Reviewing a little bit of history to remember why two lines of development were touching the same area of code can be really helpful sometimes.

To get a full list of all of the unique commits that were included in either branch involved in this merge, we can use the “triple dot” syntax that we learned in Triple Dot.

$ git log --oneline --left-right HEAD...MERGE\_HEAD

We can further simplify this though to give us much more specific context. If we add the --merge option to git log, it will only show the commits in either side of the merge that touch a file that’s currently conflicted.

$ git log --oneline --left-right –merge

If you run that with the -p option instead, you get just the diffs to the file that ended up in conflict.This can be **really** helpful in quickly giving you the context you need to help understand why something conflicts and how to more intelligently resolve it.

**Combined Diff Format**

When you run git diff directly after a merge conflict, it will give you information in a rather unique diff output format.

$ git diff

diff --cc hello.rb

index 0399cd5,59727f0..0000000

--- a/hello.rb

+++ b/hello.rb

@@@ -1,7 -1,7 +1,11 @@@

#! /usr/bin/env ruby

def hello

++<<<<<<< HEAD

+ puts 'hola world'

++=======

+ puts 'hello mundo'

++>>>>>>> mundo

end

hello()

The format is called “Combined Diff” and gives you two columns of data next to each line. The first column shows you if that line is different (added or removed) between the “ours” branch and the file in your working directory and the second column does the same between the “theirs” branch and your working directory copy.

So in that example you can see that the <<<<<<< and >>>>>>> lines are in the working copy but were not in either side of the merge. This makes sense because the merge tool stuck them in there for our context, but we’re expected to remove them.

You can also get this from the git log for any merge to see how something was resolved after the fact. Git will output this format if you run git show on a merge commit, or if you add a --cc option to a git log -p (which by default only shows patches for non-merge commits).

$ git log --cc -p -1

**Undoing Merges**

**Fix the references**

$ git reset --hard HEAD~

The downside of this approach is that it’s rewriting history, which can be problematic with a shared repository. Check out The Perils of Rebasing for more on what can happen; the short version is that if other people have the commits you’re rewriting, you should probably avoid reset. This approach also won’t work if any other commits have been created since the merge; moving the refs would effectively lose those changes.

**Reverse the commit**

If moving the branch pointers around isn’t going to work for you, Git gives you the option of making a new commit which undoes all the changes from an existing one. Git calls this operation a “revert”, and in this particular scenario, you’d invoke it like this:

$ git revert -m 1 HEAD

The -m 1 flag indicates which parent is the “mainline” and should be kept.

**Other Types of Merges**

**Our or Theirs Preference**

First of all, there is another useful thing we can do with the normal “recursive” mode of merging. We’ve already seen the ignore-all-space and ignore-space-change options which are passed with a -X but we can also tell Git to favor one side or the other when it sees a conflict.

By default, when Git sees a conflict between two branches being merged, it will add merge conflict markers into your code and mark the file as conflicted and let you resolve it. If you would prefer for Git to simply choose a specific side and ignore the other side instead of letting you manually resolve the conflict, you can pass the merge command either a -Xours or -Xtheirs

Si vous preferez que Git choisisse simplement un cote specifique et qu’il ignore l’autre cote au lieu de vous laisser fusionner manuellement le conflit, vous pouvez passer -Xours ou -Xtheirs a la commande merge.

If Git sees this, it will not add conflict markers. Any differences that are mergeable, it will merge. Any differences that conflict, it will simply choose the side you specify in whole, including binary files.

$ git merge -Xours mundo

This option can also be passed to the git merge-file command we saw earlier by running something like git merge-file --ours for individual file merges

If you want to do something like this but not have Git even try to merge changes from the other side in, there is a more draconian option, which is the “ours” merge *strategy*. This is different from the “ours” recursive merge *option*.

This will basically do a fake merge. It will record a new merge commit with both branches as parents, but it will not even look at the branch you’re merging in. It will simply record as the result of the merge the exact code in your current branch.

$ git merge -s ours mundo

$ git diff HEAD HEAD~

$

You can see that there is no difference between the branch we were on and the result of the merge

**Rerere**

The git rerere functionality is a bit of a hidden feature. The name stands for “reuse recorded resolution” and, as the name implies, it allows you to ask Git to remember how you’ve resolved a hunk conflict so that the next time it sees the same conflict, Git can resolve it for you automatically

**Debugging with Git**

**File Annotation**

you can annotate the file with git blame to determine which commit was responsible for the introduction of that line.

The following example uses git blame to determine which commit and committer was responsible for lines in the top-level Linux kernel Makefile and, further, uses the -L option to restrict the output of the annotation to lines 69 through 82 of that file:

$ git blame -L 69,82 Makefile

Notice that the first field is the partial SHA-1 of the commit that last modified that line. The next two fields are values extracted from that commit — the author name and the authored date of that commit — so you can easily see who modified that line and when. After that come the line number and the content of the file. Also note the ^1da177e4c3f4 commit lines, where the ^ prefix designates lines that were introduced in the repository’s initial commit and have remained unchanged ever since. This is a tad confusing, because now you’ve seen at least three different ways that Git uses the ^ to modify a commit SHA-1, but that is what it means here.

Another cool thing about Git is that it doesn’t track file renames explicitly. It records the snapshots and then tries to figure out what was renamed implicitly, after the fact. One of the interesting features of this is that you can ask it to figure out all sorts of code movement as well. If you pass –C to git blame, Git analyzes the file you’re annotating and tries to figure out where snippets of code within it originally came from if they were copied from elsewhere. For example, say you are refactoring a file named GITServerHandler.m into multiple files, one of which is GITPackUpload.m. By blaming GITPackUpload.m with the -C option, you can see where sections of the code originally came from:

$ git blame -C -L 141,153 GITPackUpload.m

**Submodules**

Submodules allow you to keep a Git repository as a subdirectory of another Git repository. This lets you clone another repository into your project and keep your commits separate.

**Starting with Submodules**

To add a new submodule you use the git submodule add command with the absolute or relative URL of the project you would like to start tracking

$ git submodule add <https://github.com/chaconinc/DbConnector>

If you run git status at this point, you’ll notice a few things. First you should notice the new .gitmodules file. This is a configuration file that stores the mapping between the project’s URL and the local subdirectory you’ve pulled it into

$ cat .gitmodules

[submodule "DbConnector"]

path = DbConnector

url = <https://github.com/chaconinc/DbConnector>

If you have multiple submodules, you’ll have multiple entries in this file. It’s important to note that this file is version-controlled with your other files, like your .gitignore file. It’s pushed and pulled with the rest of your project. This is how other people who clone this project know where to get the submodule projects from.

$ git diff DbConnector

Although DbConnector is a subdirectory in your working directory, Git sees it as a submodule and doesn’t track its contents when you’re not in that directory. Instead, Git sees it as a particular commit from that repository. If you want a little nicer diff output, you can pass the --submodule option to git diff.

$ git diff –submodule

**Cloning a Project with Submodules**

When you clone such a project, by default you get the directories that contain submodules, but none of the files within them yet.

The DbConnector directory is there, but empty. You must run two commands: git submodule init to initialize your local configuration file, and git submodule update to fetch all the data from that project and check out the appropriate commit listed in your superproject.

$ git submodule init

$ git submodule update

$ git submodule init

Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path 'DbConnector'

$ git submodule update

There is another way to do this which is a little simpler, however. If you pass --recurse-submodules to the git clone command, it will automatically initialize and update each submodule in the repository.

**Working on a Project with Submodules**

**Pulling in Upstream Changes**

If you want to check for new work in a submodule, you can go into the directory and run git fetch and git merge the upstream branch to update the local code.

$ git fetch

$ git merge origin/master

Now if you go back into the main project and run git diff --submodule you can see that the submodule was updated and get a list of commits that were added to it. If you don’t want to type --submodule every time you run git diff, you can set it as the default format by setting the diff.submodule config value to “log”.

$ git config --global diff.submodule log

$ git diff

There is an easier way to do this as well, if you prefer to not manually fetch and merge in the subdirectory. If you run git submodule update --remote, Git will go into your submodules and fetch and update for you.

$ git submodule update --remote DbConnector

This command will by default assume that you want to update the checkout to the master branch of the submodule repository. You can, however, set this to something different if you want. For example, if you want to have the DbConnector submodule track that repository’s “stable” branch, you can set it in either your .gitmodules file (so everyone else also tracks it), or just in your local .git/config file. Let’s set it in the .gitmodules file:

$ git config -f .gitmodules submodule.DbConnector.branch stable

$ git submodule update --remote

If you leave off the -f .gitmodules it will only make the change for you, but it probably makes more sense to track that information with the repository so everyone else does as well.

If you set the configuration setting status.submodulesummary, Git will also show you a short summary of changes to your submodules:

$ git config status.submodulesummary 1

C’est une information interessante car vous pouvez voir le journal des modifications

que vous vous appretez a valider dans votre sous-module. Une fois validees, vous pouvez encore visualiser cette information en lancant git log -p.

$ git log -p –submodule

Git will by default try to update **all** of your submodules when you run git submodule update --remote so if you have a lot of them, you may want to pass the name of just the submodule you want to try to update.

$ git submodule update --remote DbConnector

**Working on a Submodule**

So far, when we’ve run the git submodule update command to fetch changes from the submodule repositories, Git would get the changes and update the files in the subdirectory but will leave the sub-repository in what’s called a “detached HEAD” state. This means that there is no local working branch (like “master”, for example) tracking changes

First of all, let’s go into our submodule directory and check out a branch.

$ git checkout stable

Let’s try it with the “merge” option. To specify it manually, we can just add the --merge or --rebase option to our update call. Here we’ll see that there was a change on the server for this submodule and it gets merged in.

$ git submodule update --remote --merge

If we go into the DbConnector directory, we have the new changes already merged into our local stable branch. Now let’s see what happens when we make our own local change to the library and someone else pushes another change upstream at the same time.

Now if we update our submodule we can see what happens when we have made a local change and upstream also has a change we need to incorporate.

$ git submodule update --remote --rebase

If you forget the --rebase or --merge, Git will just update the submodule to whatever is on the server and reset your project to a detached HEAD state.

If you haven’t committed your changes in your submodule and you run a submodule update that would cause issues, Git will fetch the changes but not overwrite unsaved work in your submodule directory.

If you made changes that conflict with something changed upstream, Git will let you know when you run the update. You can go into the submodule directory and fix the conflict just as you normally would.

**Publishing Submodule Changes**

In order to make sure this doesn’t happen, you can ask Git to check that all your submodules have been pushed properly before pushing the main project. The git push command takes the –recurse -submodules argument which can be set to either “check” or “on-demand”. The “check” option will make push simply fail if any of the committed submodule changes haven’t been pushed.

If you want the check behavior to happen for all pushes, you can make this behavior the default by doing git config push.recurseSubmodules check

The other option is to use the “on-demand” value, which will try to do this for you.

$ git push --recurse-submodules=on-demand

As you can see there, Git went into the DbConnector module and pushed it before pushing the main project. If that submodule push fails for some reason, the main project push will also fail. You can make this behavior the default by doing git config push.recurseSubmodules on-demand.

**Submodule Tips**

There are a few things you can do to make working with submodules a little easier.

**Submodule foreach**

There is a foreach submodule command to run some arbitrary command in each submodule. This can be really helpful if you have a number of submodules in the same project.

We can easily stash all the work in all our submodules

$ git submodule foreach git stash

Then we can create a new branch and switch to it in all our submodules.

$ git submodule foreach 'git checkout -b featureA'

$ git diff; git submodule foreach 'git diff'

**Useful Aliases**

You may want to set up some aliases for some of these commands as they can be quite long and you can’t set configuration options for most of them to make them defaults.

$ git config alias.sdiff '!'"git diff && git submodule foreach 'git diff'"

$ git config alias.spush 'push --recurse-submodules=on-demand'

$ git config alias.supdate 'submodule update --remote --merge'

This way you can simply run git supdate when you want to update your submodules, or git spush to push with submodule dependency checking.

# 8.Customizing Git

**Git Configuration**

As you read briefly in Getting Started, you can specify Git configuration settings with the git config command. One of the first things you did was set up your name and email address:

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

First, a quick review: Git uses a series of configuration files to determine non-default behavior that you may want. The first place Git looks for these values is in the system-wide /etc/gitconfig file, which contains settings that are applied to every user on the system and all of their repositories. If you pass the option --system to git config, it reads and writes from this file specifically.

The next place Git looks is the ~/.gitconfig (or ~/.config/git/config) file, which is specific to each user. You can make Git read and write to this file by passing the --global option.

Finally, Git looks for configuration values in the configuration file in the Git directory (.git/config) of whatever repository you’re currently using. These values are specific to that single repository, and represent passing the --local option to git config. (If you don’t specify which level you want to work with, this is the default.)

Each of these “levels” (system, global, local) overwrites values in the previous level, so values in .git/config trump those in /etc/gitconfig, for instance

core.editor:

By default, Git uses whatever you’ve set as your default text editor via one of the shell environment variables VISUAL or EDITOR, or else falls back to the vi editor to create and edit your commit and tag messages. To change that default to something else, you can use the core.editor setting:

$ git config --global core.editor emacs

commit.template:

If you set this to the path of a file on your system, Git will use that file as the default initial message when you commit.

$ git config --global commit.template ~/.gitmessage.txt

core.autocrlf

$ git config --global core.autocrlf true

$ git config --global core.autocrlf input

If you’re a Windows programmer doing a Windows-only project, then you can turn off this functionality, recording the carriage returns in the repository by setting the config value to false:

$ git config --global core.autocrlf false

# 10 Git Internals

**Plumbing and Porcelain**

When you run git init in a new or existing directory, Git creates the .git directory, which is where almost everything that Git stores and manipulates is located.

a newlyinitialized .git directory typically looks like:

$ ll -la .git

total 11

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 ./

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 ../

-rw-r--r-- 1 asaki 1049089 130 Sep 27 18:37 config

-rw-r--r-- 1 asaki 1049089 73 Sep 27 18:37 description

-rw-r--r-- 1 asaki 1049089 23 Sep 27 18:37 HEAD

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 hooks/

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 info/

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 objects/

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 refs/

$ ls -F1

config

description

HEAD

hooks/

info/

objects/

refs/

Depending on your version of Git, you may see some additional content there, but this is a fresh git init repository — it’s what you see by default. The description file is used only by the GitWeb program, so don’t worry about it. The config file contains your project-specific configuration options, and the info directory keeps a global exclude file for ignored patterns that you don’t want to track in a .gitignore file. The hooks directory contains your client- or server-side hook scripts, which are discussed in detail in Git Hooks.

This leaves four important entries: the HEAD and (yet to be created) index files, and the objects and refs directories. These are the core parts of Git. The objects directory stores all the content for your database, the refs directory stores pointers into commit objects in that data (branches, tags,

remotes and more), the HEAD file points to the branch you currently have checked out, and the index file is where Git stores your staging area information. You’ll now look at each of these sections in detail to see how Git operates.

**Git Objects**

Git is a content-addressable filesystem. Great. What does that mean? It means that at the core of Git is a simple key-value data store. What this means is that you can insert any kind of content into a Git repository, for which Git will hand you back a unique key you can use later to retrieve that content.

$ find .git/objects

.git/objects

.git/objects/info

.git/objects/pack

$ find .git/objects -type f

Git has initialized the objects directory and created pack and info subdirectories in it, but there are no regular files. Now, let’s use git hash-object to create a new data object and manually store it in your new Git database

$ echo 'test content' | git hash-object -w –stdin

In its simplest form, git hash-object would take the content you handed to it and merely return the unique key that *would* be used to store it in your Git database. The -w option then tells the command to not simply return the key, but to write that object to the database. Finally, the --stdin option tells git hash-object to get the content to be processed from stdin; otherwise, the command would expect a filename argument at the end of the command containing the content to be used.

The output from the above command is a 40-character checksum hash. This is the SHA-1 hash — a checksum of the content you’re storing plus a header, which you’ll learn about in a bit. Now you can see how Git has stored your data:

$ find .git/objects -type f

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4

This is how Git stores the content initially — as a single file per piece of content, named with the SHA-1 checksum of the content and its header. The subdirectory is named with the first 2 characters of the SHA-1, and the filename is the remaining 38 characters.

Once you have content in your object database, you can examine that content with the git cat-file command. This command is sort of a Swiss army knife for inspecting Git objects. Passing -p to catfile instructs the command to first figure out the type of content, then display it appropriately:

$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4

test content

Now, you can add content to Git and pull it back out again. You can also do this with content in files. For example, you can do some simple version control on a file. First, create a new file and save its contents in your database:

$ echo 'version 1' > test.txt

$ git hash-object -w test.txt

83baae61804e65cc73a7201a7252750c76066a30

Then, write some new content to the file, and save it again:

$ echo 'version 2' > test.txt

$ git hash-object -w test.txt

1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

Your object database now contains both versions of this new file (as well as the first content you stored there):

$ find .git/objects -type f

.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a

.git/objects/83/baae61804e65cc73a7201a7252750c76066a30

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4

You can have Git tell you the object type of any object in Git, given its SHA-1 key, with git cat-file -t:

$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

Blob

**Tree Objects**

The next type of Git object we’ll examine is the *tree*, which solves the problem of storing the filename and also allows you to store a group of files together. Git stores content in a manner similar to a UNIX filesystem, but a bit simplified. All the content is stored as tree and blob objects, with trees corresponding to UNIX directory entries and blobs corresponding more or less to inodes or file contents. A single tree object contains one or more entries, each of which is the SHA-1 hash of a blob or subtree with its associated mode, type, and filename. For example, the most recent tree in a project may look something like this:

$ git cat-file -p master^{tree}

100644 blob a906cb2a4a904a152e80877d4088654daad0c859 README

100644 blob 8f94139338f9404f26296befa88755fc2598c289 Rakefile

040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0 lib

The master^{tree} syntax specifies the tree object that is pointed to by the last commit on your master branch. Notice that the lib subdirectory isn’t a blob but a pointer to another tree:

**Commit Objects**

This is essentially what Git does when you run the git add and git commit

commands — it stores blobs for the files that have changed, updates the index, writes out trees, and writes commit objects that reference the top-level trees and the commits that came immediately before them. These three main Git objects — the blob, the tree, and the commit — are initially stored as separate files in your .git/objects directory. Here are all the objects in the example directory now, commented with what they store:

**Object Storage**

**Git References**

Instead, itwould be easier if you had a file in which you could store that SHA-1 value under a simple name so you could use that simple name rather than the raw SHA-1 value.

In Git, these simple names are called “references” or “refs”; you can find the files that contain those SHA-1 values in the .git/refs directory. In the current project, this directory contains no files, but it does contain a simple structure:

$ find .git/refs

.git/refs

.git/refs/heads

.git/refs/tags

$ find .git/refs -type f

That’s basically what a branch in Git is: a simple pointer or reference to the head of a line of work.

**The HEAD**

The HEAD file is a symbolic reference to the branch you’re currently on. By symbolic reference, we mean that unlike a normal reference, it doesn’t generally contain a SHA-1 value but rather a pointer to another reference.

$ cat .git/HEAD

ref: refs/heads/master

When you run git commit, it creates the commit object, specifying the parent of that commit object to be whatever SHA-1 value the reference in HEAD points to.

**Tags**

We just finished discussing Git’s three main object types (*blobs*, *trees* and *commits*), but there is a fourth. The *tag* object is very much like a commit object — it contains a tagger, a date, a message, and a pointer. The main difference is that a tag object generally points to a commit rather than a tree. It’s like a branch reference, but it never moves — it always points to the same commit but gives it a friendlier name

That is all a lightweight tag is — a reference that never moves. An annotated tag is more complex, however. If you create an annotated tag, Git creates a tag object and then writes a reference to point to it rather than directly to the commit. You can see this by creating an annotated tag (using the -a option):

$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'

Here’s the object SHA-1 value it created:

$ cat .git/refs/tags/v1.1

9585191f37f7b0fb9444f35a9bf50de191beadc2

**Remotes**

The third type of reference that you’ll see is a remote reference. If you add a remote and push to it, Git stores the value you last pushed to that remote for each branch in the refs/remotes directory

Then, you can see what the master branch on the origin remote was the last time you

communicated with the server, by checking the refs/remotes/origin/master file:

$ cat .git/refs/remotes/origin/master

ca82a6dff817ec66f44342007202690a93763949

Remote references differ from branches (refs/heads references) mainly in that they’re considered read-only. You can git checkout to one, but Git won’t point HEAD at one, so you’ll never update it with a commit command. Git manages them as bookmarks to the last known state of where those branches were on those servers.

**Packfiles**

The initial format in which Git saves objects on disk is called a “loose” object

format. However, occasionally Git packs up several of these objects into a single binary file called a “packfile” in order to save space and be more efficient. Git does this if you have too many loose objects around, if you run the git gc command manually

If you look in your objects directory, you’ll find that most of your objects are gone, and a new pair of files has appeared:

$ find .git/objects -type f

.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4

.git/objects/info/packs

.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx

.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack

The objects that remain are the blobs that aren’t pointed to by any commit — in this case, the “what is up, doc?” example and the “test content” example blobs you created earlier. Because you never added them to any commits, they’re considered dangling and aren’t packed up in your new packfile.

The other files are your new packfile and an index. The packfile is a single file containing the contents of all the objects that were removed from your filesystem. The index is a file that contains offsets into that packfile so you can quickly seek to a specific object. What is cool is that although the objects on disk before you ran the gc command were collectively about 15K in size, the new packfile is only 7K. You’ve cut your disk usage by half by packing your objects.

How does Git do this? When Git packs objects, it looks for files that are named and sized similarly, and stores just the deltas from one version of the file to the next. You can look into the packfile and see what Git did to save space. The git verify-pack plumbing command allows you to see what was packed up:

$ git verify-pack -v .git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx

**The Refspec**

$ git remote add origin https://github.com/schacon/simplegit-progit

Running the command above adds a section to your repository’s .git/config file, specifying the name of the remote (origin), the URL of the remote repository, and the *refspec* to be used for fetching:

[remote "origin"]

url = https://github.com/schacon/simplegit-progit

fetch = +refs/heads/\*:refs/remotes/origin/\*

The format of the refspec is, first, an optional +, followed by <src>:<dst>, where <src> is the pattern for references on the remote side and <dst> is where those references will be tracked locally. The + tells Git to update the reference even if it isn’t a fast-forward.In the default case that is automatically written by a git remote add command, Git fetches all the references under refs/heads/ on the server and writes them to refs/remotes/origin/ locally. So, if there is a master branch on the server, you can access the log of that branch locally via any of the following:

$ git log origin/master

$ git log remotes/origin/master

$ git log refs/remotes/origin/master

They’re all equivalent, because Git expands each of them to refs/remotes/origin/master.

If you want Git instead to pull down only the master branch each time, and not every other branch on the remote server, you can change the fetch line to refer to that branch only:

fetch = +refs/heads/master:refs/remotes/origin/master

This is just the default refspec for git fetch for that remote. If you want to do a one-time only fetch, you can specify the specific refspec on the command line, too. To pull the master branch on the remote down to origin/mymaster locally, you can run:

$ git fetch origin master:refs/remotes/origin/mymaster

You can also specify multiple refspecs for fetching in your configuration file. If you want to always fetch the master and experiment branches from the origin remote, add two lines:

[remote "origin"]

url = https://github.com/schacon/simplegit-progit

fetch = +refs/heads/master:refs/remotes/origin/master

fetch = +refs/heads/experiment:refs/remotes/origin/experiment

However, you can use namespaces (or directories) to accomplish something like that. If you have a QA team that pushes a series of branches, and you want to get the master branch and any of the QA team’s branches but nothing else, you can use a config section like this:

[remote "origin"]

url = https://github.com/schacon/simplegit-progit

fetch = +refs/heads/master:refs/remotes/origin/master

fetch = +refs/heads/qa/\*:refs/remotes/origin/qa/\*

If you have a complex workflow process that has a QA team pushing branches, developers pushing branches, and integration teams pushing and collaborating on remote branches, you can namespace them easily this way.

**Pushing Refspecs**

It’s nice that you can fetch namespaced references that way, but how does the QA team get their branches into a qa/ namespace in the first place? You accomplish that by using refspecs to push. If the QA team wants to push their master branch to qa/master on the remote server, they can run

$ git push origin master:refs/heads/qa/master

If they want Git to do that automatically each time they run git push origin, they can add a push value to their config file:

[remote "origin"]

url = https://github.com/schacon/simplegit-progit

fetch = +refs/heads/\*:refs/remotes/origin/\*

push = refs/heads/master:refs/heads/qa/master

Again, this will cause a git push origin to push the local master branch to the remote qa/master branch by default.

**Deleting References**

You can also use the refspec to delete references from the remote server by running something like this:

$ git push origin :topic

Because the refspec is <src>:<dst>, by leaving off the <src> part, this basically says to make the topic branch on the remote nothing, which deletes it.

Or you can use the newer syntax (available since Git v1.7.0):

$ git push origin --delete topic

**Maintenance and Data Recovery**

Occasionally, you may have to do some cleanup – make a repository more compact, clean up an imported repository, or recover lost work. This section will cover some of these scenarios.

**Maintenance**

Occasionally, Git automatically runs a command called “auto gc”. Most of the time, this command does nothing. However, if there are too many loose objects (objects not in a packfile) or too many packfiles, Git launches a full-fledged git gc command. The “gc” stands for garbage collect, and the command does a number of things: it gathers up all the loose objects and places them in packfiles, it consolidates packfiles into one big packfile, and it removes objects that aren’t reachable from any commit and are a few months old.

$ git gc

**Data Recovery**

Often, the quickest way is to use a tool called git reflog. As you’re working, Git silently records what your HEAD is every time you change it. Each time you commit or change branches, the reflog is updated. The reflog is also updated by the git update-ref command, which is another reason to use it instead of just writing the SHA-1 value to your ref files, as we covered in Git References. You can see where you’ve been at any time by running git reflog:

$ git reflog

1a410ef HEAD@{0}: reset: moving to 1a410ef

ab1afef HEAD@{1}: commit: modified repo.rb a bit

484a592 HEAD@{2}: commit: added repo.rb

To see the same information in a much more useful way, we can run git log -g, which will give you a normal log output for your reflog

Because the reflog data is kept in the .git/logs/ directory, you effectively have no reflog. How can you recover that commit at this point? One way is to use the git fsck utility, which checks your database for integrity. If you run it with the --full option, it shows you all objects that aren’t pointed to by another object:

$ git fsck --full

Checking object directories: 100% (256/256), done.

Checking objects: 100% (18/18), done.

dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4

dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b

dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9

dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293

Vous pouvez executer la commande count-objects pour voir rapidement combien d’espace disque vous utilisez :

$ git count-objects -v

# Appendix A: Git in Other Environments

gitk is a graphical history viewer. Think of it like a powerful GUI shell over git log and git grep

Gitk is easiest to invoke from the command-line. Just cd into a Git repository, and type:

$ gitk [git log options]

Gitk accepts many command-line options, most of which are passed through to the underlying git log action. Probably one of the most useful is the --all flag, which tells gitk to show commits reachable from *any* ref, not just HEAD.

git-gui, on the other hand, is primarily a tool for crafting commits. It, too, is easiest to invoke from the command line:

$ git gui

gitk and git-gui are examples of task-oriented tools. Each of them is tailored for a specific purpose (viewing history and creating commits, respectively), and omit the features not necessary for that task.

# Appendix C: Git Commands

**Setup and Config**

**git config**

Git has a default way of doing hundreds of things. For a lot of these things, you can tell Git to default to doing them a different way, or set your preferences.

**git help**

The git help command is used to show you all the documentation shipped with Git about any command

$ git help <command>

**Getting and Creating Projects**

**git init**

To take a directory and turn it into a new Git repository so you can start version controlling it, you can simply run git init.

**git clone**

The git clone command is actually something of a wrapper around several other commands. It creates a new directory, goes into it and runs git init to make it an empty Git repository, adds a remote (git remote add) to the URL that you pass it (by default named origin), runs a git fetch from that remote repository and then checks out the latest commit into your working directory with git checkout.

the --bare option to create a copy of a Git repository with no working directory

the --recurse-submodules option to makecloning a repository with submodules a little simpler.

**Basic Snapshotting:**

For the basic workflow of staging content and committing it to your history, there are only a few basic commands.

**git add**

The git add command adds content from the working directory into the staging area (or “index”) for the next commit.

**git status**

The git status command will show you the different states of files in your working directory and staging area. Which files are modified and unstaged and which are staged but not yet committed

**git diff:**

The git diff command is used when you want to see differences between any two trees. This could be the difference between your working environment and your staging area (git diff by itself), between your staging area and your last commit (git diff --staged), or between two commits (git diff master branchB)

We use it to look for possible whitespace issues before committing with the --check option in Commit Guidelines.

We see how to check the differences between branches more effectively with the git diff A...B syntax in Determining What Is Introduced.

We use it to filter out whitespace differences with -b and how to compare different stages of conflicted files with --theirs, --ours and --base in Advanced Merging.

Finally, we use it to effectively compare submodule changes with --submodule in Starting with Submodules.

**git difftool:**

The git difftool command simply launches an external tool to show you the difference between two trees in case you want to use something other than the built in git diff command.

**git commit**

The git commit command takes all the file contents that have been staged with git add and records a new permanent snapshot in the database and then moves the branch pointer on the current branch up to it.

We first cover the basics of committing in Committing Your Changes. There we also demonstrate how to use the -a flag to skip the git add step in daily workflows and how to use the -m flag to pass a commit message in on the command line instead of firing up an editor

the --amend option to redo the most recent commit.

the -S flag sign commits cryptographically

**git reset:**

The git reset command is primarily used to undo things, as you can possibly tell by the verb. It moves around the HEAD pointer and optionally changes the index or staging area and can also

optionally change the working directory if you use --hard. This final option makes it possible for this command to lose your work if used incorrectly, so make sure you understand it before using it.

We first effectively cover the simplest use of git reset in Unstaging a Staged File, where we use it to unstage a file we had run git add on.

**git rm**

The git rm command is used to remove files from the staging area and working directory for Git

including recursively removing files and only removing files from the staging area but leaving them in the working directory with --cached.

**git mv**

The git mv command is a thin convenience command to move a file and then run git add on the new file and git rm on the old file.

**git clear**

The git mv command is a thin convenience command to move a file and then run git add on the new file and git rm on the old file.

**Branching and Merging**

There are just a handful of commands that implement most of the branching and merging

functionality in Git.

**git branch**

The git branch command is actually something of a branch management tool. It can list the branches you have, create a new branch, delete branches and rename branches

In Tracking Branches we use the git branch -u option to set up a tracking branch.

**git checkout**

The git checkout command is used to switch branches and check content out into your working directory.

We see how to use it to start tracking branches with the --track flag in Tracking Branches.

**git merge**

The git merge tool is used to merge one or more branches into the branch you have checked out. It will then advance the current branch to the result of the merge.

We use it to reintroduce file conflicts with --conflict=diff3 in Checking Out Conflicts.

We go into closer detail on its relationship with git reset in Reset Demystified.

**git mergetool**

The git mergetool command simply launches an external merge helper in case you have issues with a merge in Git.

**git log**

The git log command is used to show the reachable recorded history of a project from the most recent commit snapshot backwards.

There we look at the -p and --stat option to get an idea of what was introduced in each commit and the --pretty and --oneline options to view the history more concisely, along with some simple date and author filtering options.

we use it with the --decorate option to easily visualize where our branch

pointers are located and we also use the --graph option to see what divergent histories look like

In Private Small Team and Commit Ranges we cover the branchA..branchB syntax to use the git log command to see what commits are unique to a branch relative to another branch. In Commit Ranges we go through this fairly extensively

In Merge Log and Triple Dot we cover using the branchA...branchB format and the --left-right syntax to see what is in one branch or the other but not in both. In Merge Log we also look at how to use the --merge option to help with merge conflict debugging as well as using the --cc option to look at merge commit conflicts in your history.

In RefLog Shortnames we use the -g option to view the Git reflog through this tool instead of doing branch traversal.

In Searching we look at using the -S and -L options to do fairly sophisticated searches for something that happened historically in the code such as seeing the history of a function.

In Signing Commits we see how to use --show-signature to add a validation string to each commit in the git log output based on if it was validly signed or not.

**git stash**

The git stash command is used to temporarily store uncommitted work in order to clean out your working directory without having to commit unfinished work on a branch

**git tag**

The git tag command is used to give a permanent bookmark to a specific point in the code history. Generally this is used for things like releases. This command is introduced and covered in detail

**Sharing and Updating Projects**

**git fetch**

The git fetch command communicates with a remote repository and fetches down all the information that is in that repository that is not in your current one and stores it in your local database.

**git pull**

The git pull command is basically a combination of the git fetch and git merge commands, where Git will fetch from the remote you specify and then immediately try to merge it into the branch you’re on.

**git push**

The git push command is used to communicate with another repository, calculate what your local database has that the remote one does not, and then pushes the difference into the other repository. It requires write access to the other repository and so normally is authenticated somehow.

In Deleting Remote Branches we use the --delete flag to delete a branch on the server with git push.

We see how to use it to share tags that you have made with the --tags option in Sharing Tags.

In Publishing Submodule Changes we use the --recurse-submodules option to check that all of our submodules work has been published before pushing the superproject, which can be really helpful when using submodules.

**git remote**

The git remote command is a management tool for your record of remote repositories.

It is used in nearly every subsequent chapter in the book too, but always in the standard git remote add <name> <url> format

**git archive**

The git archive command is used to create an archive file of a specific snapshot of the project.

**git submodule**

The git submodule command is used to manage external repositories within a normal repositories. This could be for libraries or other types of shared resources. The submodule command has several sub-commands (add, update, sync, etc) for managing these resources.

**Inspection and Comparison**

**git show**

The git show command can show a Git object in a simple and human readable way. Normally you would use this to show the information about a tag or a commit.

**git shortlog**

The git shortlog command is used to summarize the output of git log. It will take many of the same options that the git log command will but instead of listing out all of the commits it will present a summary of the commits grouped by author.

**git describe**

The git describe command is used to take anything that resolves to a commit and produces a string that is somewhat human-readable and will not change. It’s a way to get a description of a commit that is as unambiguous as a commit SHA-1 but more understandable.We use git describe in Generating a Build Number and Preparing a Release to get a string to name our release file after.

**Debogage**

**git bisect**

The git bisect tool is an incredibly helpful debugging tool used to find which specific commit was the first one to introduce a bug or problem by doing an automatic binary search.

**git blame:**

The git blame command annotates the lines of any file with which commit was the last one to introduce a change to each line of the file and what person authored that commit. This is helpful in order to find the person to ask for more information about a specific section of your code.

**git grep**

The git grep command can help you find any string or regular expression in any of the files in your source code, even older versions of your project.

**Patching**

**git cherry-pick**

The git cherry-pick command is used to take the change introduced in a single Git commit and try to re-introduce it as a new commit on the branch you’re currently on. This can be useful to only take one or two commits from a branch individually rather than merging in the branch which takes all the changes.

**git rebase**

The git rebase command is basically an automated cherry-pick. It determines a series of commits and then cherry-picks them one by one in the same order somewhere else.

We also use it in an interactive scripting mode with the -i option

**git revert**

The git revert command is essentially a reverse git cherry-pick. It creates a new commit that applies the exact opposite of the change introduced in the commit you’re targeting, essentially undoing or reverting it.

**External Systems**

Git comes with a few commands to integrate with other version control systems.

**git svn**

The git svn command is used to communicate with the Subversion version control system as a client. This means you can use Git to checkout from and commit to a Subversion server.

**Administration**

**git gc**

The git gc command runs “garbage collection” on your repository, removing unnecessary files in your database and packing up the remaining files into a more efficient format.

**git fsck**

The git fsck command is used to check the internal database for problems or inconsistencies.

**git reflog**

The git reflog command goes through a log of where all the heads of your branches have been as you work to find commits you may have lost through rewriting histories.

We cover this command mainly in RefLog Shortnames, where we show normal usage to and how to use git log -g to view the same information with git log output.

We also go through a practical example of recovering such a lost branch in Data Recovery

**git filter-branch**

The git filter-branch command is used to rewrite loads of commits according to certain patterns, like removing a file everywhere or filtering the entire repository down to a single subdirectory for extracting a project

**Plumbing Commands**

ls-remote to look at the raw references on the server

ls-files to take a more raw look at what your staging area looks like.

rev-parse to take just about any string and turn it into an object SHA-1.

**Git distribue**

**Git and Other Systems**

.